ITU FRP 2010

Lecture 2: Yampa: Arrows-based FRP

Henrik Nilsson

School of Computer Science
University of Nottingham, UK
Outline

- CFRP issues
- Introduction to Yampa
- Arrows
- A closer look at Yampa
CFRP issues: Sharing

Consider:

\[
\text{let } x = 1 + \text{integral} \ (x \times x) \text{ in } x
\]

The recursively defined behavior, a \textit{function}, is applied over and over to the \textit{same} stream of sample times.

- Causes recomputation
- Laziness does \textit{not} help
- Memoization needed to get acceptable performance. But with care to avoid memory leaks.
CFRP issues: Restart (1)

Consider:

```plaintext
let
  c = hold 0 (count (repeatedly 0.5))
in
  c 'until' after 5 => c * 2
```

What happened at the time of the switch?

- CFRP behaviors and events are *signal generators*: they will start from scratch when switched in.
- But what if we just want to continue observing an evolving signal?
CFRP issues: Restart (2)

- A version of \texttt{until} that starts new behaviors from time 0.
  \textit{Time and space leak!}

- Support signals as well, e.g. through some variant of \texttt{runningIn}:

  \begin{verbatim}
  runningIn ::
  B a -> (B a -> B b) -> B b
  \end{verbatim}

  Idea: apply behavior to start time once and for all, then wrap up the resulting signal as a signal generator that ignores its starting time.
Problems with \texttt{runningIn}

- No type-level distinction between signals and signal generators: a “running behavior” is a signal masquerading as a signal generator. (But could be fixed though other designs.)
- Difficult to implement; requires imperative techniques, implies certain overhead.
An alternative

By adopting *signal functions* as the central notion, these problems are side stepped:

- Sharing amounts to sharing computations of signal samples: lazy evaluation handles that just fine.
- Observation of externally originating signals is inherent in the notion of a signal function.
- Implementation is straightforward.
Yampa

What is *Yampa*?

- FRP implementation structured using *arrows*. 
Yampa

What is *Yampa*?

- FRP implementation structured using *arrows*.
- Realised as an *Embedded Domain-Specific Language* (EDSL), i.e. a combinator library.
Yampa

What is **Yampa**?

- FRP implementation structured using **arrows**.
- Realised as an **Embedded Domain-Specific Language (EDSL)**, i.e. a combinator library.
- **Continuous-time** signals (conceptually)
What is **Yampa**?

- FRP implementation structured using *arrows*.
- Realised as an *Embedded Domain-Specific Language* (EDSL), i.e. a combinator library.
- *Continuous-time* signals (conceptually)
- Discrete-time signals represented by continuous-time signal carrying option type *Event*.
Yampa

What is **Yampa**?

- FRP implementation structured using **arrows**.
- Realised as an **Embedded Domain-Specific Language** (EDSL), i.e. a combinator library.
- **Continuous-time** signals (conceptually)
- Discrete-time signals represented by continuous-time signal carrying option type **Event**.
- Functions on signals, **Signal Functions**, is the central abstraction, forming the arrows.
- Signal functions are first-class entities, signals a secondary notion, only existing indirectly through the signal functions.
Signal functions are first-class entities, signals a secondary notion, only existing indirectly through the signal functions.

Advanced *switching constructs* to describe systems with highly dynamic structure.
• Signal functions are first-class entities, signals a secondary notion, only existing indirectly through the signal functions.

• Advanced *switching constructs* to describe systems with highly dynamic structure.

• People:
  - Antony Courtney
  - Paul Hudak
  - Henrik Nilsson
  - John Peterson
Yampa?
Yampa?

Yet
Another
Mostly
Pointless
Acronym
Yampa?

Yet
Another
Mostly
Pointless
Acronym

???
Yampa?

Yet
Another
Mostly
Pointless
Acronym

???

No …
Yampa?

Yampa is a river . . .
Yampa?

... with long calmly flowing sections ...
Yampa?

... and abrupt whitewater transitions in between.

A good metaphor for hybrid systems!
Signal functions (1)

Key concept: *functions on signals*. 

![Diagram](attachment:image.png)
Key concept: **functions on signals**.

**Intuition:**

\[
\begin{align*}
\text{Signal } \alpha & \approx \text{Time} \rightarrow \alpha \\
\text{SF } \alpha \beta & \approx \text{Signal } \alpha \rightarrow \text{Signal } \beta
\end{align*}
\]

\[
\begin{align*}
x & :: \text{Signal T1} \\
y & :: \text{Signal T2} \\
f & :: \text{SF T1 T2}
\end{align*}
\]
Signal functions (2)

Additionally, *causality* required: output at time $t$ must be determined by input on interval $[0, t]$. 
Additionally, **causality** required: output at time $t$ must be determined by input on interval $[0, t]$.

Signal functions are said to be

- **pure** or **stateless** if output at time $t$ only depends on input at time $t$. 


Signal functions (2)

Additionally, *causality* required: output at time $t$ must be determined by input on interval $[0, t]$.

Signal functions are said to be

- **pure** or **stateless** if output at time $t$ only depends on input at time $t$

- **impure** or **stateful** if output at time $t$ depends on input over the interval $[0, t]$. 
Signal functions and state

Alternative view:
Signal functions and state

Alternative view:

Signal functions can encapsulate state.

\[ \text{state}(t) \text{ summarizes input history } x(t'), \ t' \in [0, t]. \]

Thus, really a kind of process.
Signal functions and state

Alternative view:

Signal functions can encapsulate \textit{state}.

\[
\begin{align*}
\text{state}(t) \text{ summarizes input history } x(t'), \ t' \in [0, t].
\end{align*}
\]

Thus, really a kind of \textit{process}.

From this perspective, signal functions are:

- \textbf{stateful} if \( y(t) \) depends on \( x(t) \) and \textit{state}(t)
- \textbf{stateless} if \( y(t) \) depends only on \( x(t) \)
In Yampa, systems are described by combining signal functions (forming new signal functions).
Yampa and arrows (1)

In Yampa, systems are described by combining signal functions (forming new signal functions).

For example, serial composition:

\[
\begin{array}{c}
\begin{array}{c}
 f \\
\end{array}
\end{array}
\xrightarrow{\rightarrow}
\begin{array}{c}
\begin{array}{c}
 g \\
\end{array}
\end{array}
\]
Yampa and arrows (1)

In Yampa, systems are described by combining signal functions (forming new signal functions).

For example, serial composition:

A *combinator* can be defined that captures this idea:

\[(\gggg) :: SF \ a \ b \rightarrow SF \ b \ c \rightarrow SF \ a \ c\]
But systems can be complex:
Yampa and arrows (2)

But systems can be complex:

How many and what combinators do we need to be able to describe arbitrary systems?
John Hughes’ *arrow* framework:

- Abstract data type interface for function-like types.
Yampa and arrows (3)

John Hughes’ *arrow* framework:

- Abstract data type interface for function-like types.
- Particularly suitable for types representing process-like computations.
John Hughes’ **arrow** framework:

- Abstract data type interface for function-like types.
- Particularly suitable for types representing process-like computations.
- Related to **monads**, since arrows are (effectful) computations, but more general: any monad $m$ induces an arrow, the Kleisli arrow, $\alpha \rightarrow m \beta$, but not vice versa.
John Hughes’ *arrow* framework:

- Abstract data type interface for function-like types.
- Particularly suitable for types representing process-like computations.
- Related to *monads*, since arrows are (effectful) computations, but more general: any monad $m$ induces an arrow, the Kleisli arrow, $\alpha \rightarrow m \beta$, but not vice versa.
- Provides a minimal set of “wiring” combinators.
What is an arrow? (1)

- A type constructor $a$ of arity two.
What is an arrow? (1)

- A *type constructor* \( a \) of arity two.
- Three operators:
What is an arrow? (1)

- A *type constructor* `a` of arity two.
- Three operators:
  - *lifting*:
    ```
    arr :: (b->c) -> a b c
    ```
What is an arrow? (1)

- A **type constructor** `a` of arity two.
- Three operators:
  - **lifting**:
    \[
    \text{arr} :: (b \to c) \to a \ b \ c
    \]
  - **composition**:
    \[
    (\gg\gg) :: a \ b \ c \to a \ c \ d \to a \ b \ d
    \]
What is an arrow? (1)

- A **type constructor** of arity two.
- Three operators:
  - **lifting**:  
    \[
    \text{arr} :: (b \to c) \to a \times b \times c
    \]
  - **composition**:  
    \[
    (\gg\gg\gg) :: a \times b \times c \to a \times c \times d \to a \times b \times d
    \]
  - **widening**:  
    \[
    \text{first} :: a \times b \times c \to a \times (b, d) \times (c, d)
    \]
What is an arrow? (1)

- A **type constructor** `a` of arity two.
- Three operators:
  - **lifting**:
    \[
    \text{arr} :: (b \rightarrow c) \rightarrow a \ b \ c
    \]
  - **composition**:
    \[
    (\ggg) :: a \ b \ c \rightarrow a \ c \ d \rightarrow a \ b \ d
    \]
  - **widening**:
    \[
    \text{first} :: a \ b \ c \rightarrow a \ (b,d) \ (c,d)
    \]
- A set of **algebraic laws** that must hold.
What is an arrow? (2)

These diagrams convey the general idea:

[arr] f

f >>> g

[first] f
The **Arrow** class

In Haskell, a *type class* is used to capture these ideas (except for the laws):

```haskell
class Arrow a where
    arr :: (b -> c) -> a b c
    (<<<) :: a b c -> a c d -> a b d
    first :: a b c -> a (b,d) (c,d)
```
Arrow laws

\[(f >>> g) >>> h = f >>> (g >>> h)\]
Arrow laws

\[(f >>> g) >>> h = f >>> (g >>> h)\]

\[\text{arr } (g . f) = \text{arr } f >>> \text{arr } g\]
Arrow laws

\[(f >>> g) >>> h = f >>> (g >>> h)\]

\[\text{arr } (g \ . \ f) = \text{arr } f >>> \text{arr } g\]

\[\text{arr } \text{id} >>> f = f\]
Arrow laws

\[(f >>> g) >>> h = f >>> (g >>> h)\]
\[\text{arr} \ (g \ . \ f) = \text{arr} \ f >>> \text{arr} \ g\]
\[\text{arr} \ \text{id} >>> f = f\]
\[f = f >>> \text{arr} \ \text{id}\]
Arrow laws

(f >>> g) >>> h = f >>> (g >>> h)

arr (g . f) = arr f >>> arr g

arr id >>> f = f

f = f >>> arr id

first (arr f) = arr (f \times id)
Arrow laws

\[(f >>> g) >>> h = f >>> (g >>> h)\]

\[\text{arr } (g \cdot f) = \text{arr } f >>> \text{arr } g\]

\[\text{arr id} >>> f = f\]

\[f = f >>> \text{arr id}\]

\[\text{first } (\text{arr } f) = \text{arr } (f \times \text{id})\]

\[\text{first } (f >>> g) = \text{first } f >>> \text{first } g\]
Arrow laws

\[(f >>> g) >>> h = f >>> (g >>> h)\]

\[\text{arr} (g \cdot f) = \text{arr} f >>> \text{arr} g\]

\[\text{arr} \ id >>> f = f\]

\[f = f >>> \text{arr} \ id\]

\[\text{first} \ (\text{arr} f) = \text{arr} \ (f \times \ id)\]

\[\text{first} \ (f >>> g) = \text{first} \ f >>> \text{first} \ g\]

\[\text{first} \ f >>> \text{arr} \ (\id \times g) = \text{arr} \ (\id \times g) >>> \text{first} \ f\]
Arrow laws

\[(f >>> g) >>> h = f >>> (g >>> h)\]

\[\text{arr } (g . f) = \text{arr } f >>> \text{arr } g\]

\[\text{arr id} >>> f = f\]

\[f = f >>> \text{arr id}\]

\[\text{first } (\text{arr } f) = \text{arr } (f \times \text{id})\]

\[\text{first } (f >>> g) = \text{first } f >>> \text{first } g\]

\[\text{first } f >>> \text{arr } (\text{id} \times g) = \text{arr } (\text{id} \times g) >>> \text{first } f\]

\[\text{first } f >>> \text{arr } \text{fst} = \text{arr } \text{fst} >>> f\]
Arrow laws

\[(f >>> g) >>> h = f >>> (g >>> h)\]
\[\text{arr } (g . f) = \text{arr } f >>> \text{arr } g\]
\[\text{arr } \text{id} >>> f = f\]
\[f = f >>> \text{arr } \text{id}\]
\[\text{first } (\text{arr } f) = \text{arr } (f \times \text{id})\]
\[\text{first } (f >>> g) = \text{first } f >>> \text{first } g\]
\[\text{first } f >>> \text{arr } (\text{id} \times g) = \text{arr } (\text{id} \times g) >>> \text{first } f\]
\[\text{first } f >>> \text{arr } \text{fst} = \text{arr } \text{fst} >>> f\]
\[\text{first } (\text{first } f) >>> \text{arr } \text{assoc} = \text{arr } \text{assoc} >>> \text{first } f\]
Functions are arrows (1)

Functions are a simple example of arrows. The arrow type constructor is just \((\to)\) in that case.

**Exercise 1:** Suggest suitable definitions of

- `arr`
- `>>>(>>>)`
- `first`

for this case!
Functions are arrows (2)

Solution:

- \texttt{arr = id}
Solution:

- \( \text{arr} = \text{id} \)

To see this, recall

\[
\text{id} :: t \rightarrow t \\
\text{arr} :: (b \rightarrow c) \rightarrow a \ b \ c
\]
Functions are arrows (2)

Solution:

- \text{arr} = \text{id}

To see this, recall

\[
\begin{align*}
\text{id} & : : t \rightarrow t \\
\text{arr} & : : (b \rightarrow c) \rightarrow a \ b \ c
\end{align*}
\]

Instantiate with

\[
\begin{align*}
a & = (\rightarrow) \\
t & = b \rightarrow c = (\rightarrow) \ b \ c
\end{align*}
\]
Functions are arrows (3)

- \( f >>> g = \lambda a \rightarrow g (f a) \)
Functions are arrows (3)

- \( f >>> g = \lambda a \rightarrow g (f a) \)
- \( f >>> g = g \cdot f \)
Functions are arrows (3)

- $f >>> g = \lambda a \rightarrow g \ (f \ a)$  
  or
- $f >>> g = g \ . \ f$  
  or even
- $(>>>) = \text{flip} \ (.\)$
Functions are arrows (3)

- \( f >>> g = \lambda a \to g (f a) \) or
- \( f >>> g = g \cdot f \) or even
- \((>>>) = flip (.)\)
- first \( f = \lambda (b,d) \to (f b,d) \)
Functions are arrows (4)

Arrow instance declaration for functions:

```haskell
instance Arrow (->) where
  arr      = id
  (>>>>)   = flip (.)
  first f = \(b,d) -> (f b,d)
```
The arrow laws reformulated

Exploiting that functions are arrows, some of the laws can be formulated more neatly. E.g:

\[
\begin{align*}
arr (f >>> g) &= arr f >>> arr g \\
first (arr f) &= arr (first f)
\end{align*}
\]
Another important operator is \texttt{loop}: a fixed-point operator used to express recursive arrows or \texttt{feedback}:
The `loop` combinator (2)

Not all arrow instances support `loop`. It is thus a method of a separate class:

```haskell
class Arrow a => ArrowLoop a where
  loop :: a (b, d) (c, d) -> a b c
```

Remarkably, the four combinators `arr`, `>>>`, `first`, and `loop` are sufficient to express any conceivable wiring!
Some more arrow combinators (1)

second :: Arrow a =>
           a b c -> a (d,b) (d,c)

(*** ) :: Arrow a =>
           a b c -> a d e -> a (b,d) (c,e)

(&&&) :: Arrow a =>
          a b c -> a b d -> a b (c,d)
Some more arrow combinators (2)

As diagrams:

second $f$

$f$ *** $g$

$f$ & & & $g$
Some more arrow combinators (3)

**Exercise 2:** Describe the following circuit using arrow combinators:

```
  a1 -> a2 -> a3
  |        +
  |        |
  |        |
  |        |
```

a1, a2, a3 :: A Double Double

**Exercise 3:** The combinators second, (** * **), and (&&& ) are not primitive, but defined in terms of arr, (>>>) , and first. Suggest suitable definitions!
Exercise 2: Describe the following circuit using arrow combinators:

$\text{a1, a2, a3 :: A Double Double}$
Exercise 2: Describe the following circuit using arrow combinators:

\[ a1, a2, a3 :: A \text{ Double Double} \]
\[ \text{circuit}_v1 :: A \text{ Double Double} \]
\[ \text{circuit}_v1 = (a1 &&& arr \text{ id}) >>> (a2 *** a3) >>> arr (uncurry (+)) \]
Exercise 2: Describe the following circuit:

\[ a_1, a_2, a_3 :: \text{A Double Double} \]
Exercise 2: Another solution

Exercise 2: Describe the following circuit:

\[ a_1, a_2, a_3 :: \text{A Double Double} \]

\[ \text{circuit\_v2} :: \text{A Double Double} \]

\[ \text{circuit\_v2} = \text{arr} (\lambda x \to (x,x)) \]

\[ \ggg \ggg \text{first } a_1 \]

\[ \ggg \ggg (a_2 * * * a_3) \]

\[ \ggg \ggg \text{arr (uncurry (+))} \]
Exercise 3: Suggest definitions of second, (** * ** ), and (& & & ).
**Exercise 3:** Suggest definitions of `second`, `***`, and `&&&`.

```haskell
second :: Arrow a => a b c -> a (d,b) (d,c)
second f = arr swap >>> first f >>> arr swap
swap (x,y) = (y,x)
```
Exercise 3: Suggest definitions of second, (***) and (&&&).

second :: Arrow a => a b c -> a (d,b) (d,c)
second f = arr swap >>> first f >>> arr swap
swap (x,y) = (y,x)

(***) :: Arrow a =>
    a b c -> a d e -> a (b,d) (c,e)
f *** g = first f >>> second g
Exercise 3: Suggest definitions of \texttt{second}, \texttt{(* * *)}, and \texttt{(& & &)}.

\texttt{second} :: \texttt{Arrow a => a b c \rightarrow a (d,b) (d,c)}

\texttt{second f = arr swap >>> first f >>> arr swap}

\texttt{swap (x,y) = (y,x)}

\texttt{(* * *)} :: \texttt{Arrow a => \rightarrow a b c \rightarrow a (b,d) (c,e)}

\texttt{f *** g = first f >>> second g}

\texttt{(& & &)} :: \texttt{Arrow a => a b c \rightarrow a b d \rightarrow a b (c,d)}

\texttt{f &&& g = arr (\lambda x \rightarrow (x,x)) >>> (f *** g)}
Note on the definition of (***

Are the following two definitions of (***

1. \( f \ *** \ g = \text{first } f \ >>>> \text{second } g \) 
2. \( f \ *** \ g = \text{second } g \ >>>> \text{first } f \)
Note on the definition of (***) (1)

Are the following two definitions of (***
) equivalent?

• $f *** g = \text{first } f >>> \text{second } g$
• $f *** g = \text{second } g >>> \text{first } f$

No, in general

$\text{first } f >>> \text{second } g \neq \text{second } g >>> \text{first } f$

since the order of the two possibly effectful computations $f$ and $g$ are different.
Similarly

\[(f \ast \ast \ast g) >>> (h \ast \ast \ast k) \neq (f >>> h) \ast \ast \ast (g >>> k)\]

since the order of \(f\) and \(g\) differs.
Note on the definition of \((***)\) (2)

Similarly

\[(f *** g) >>> (h *** k) \neq (f >>> h) *** (g >>> k)\]

since the order of \(f\) and \(g\) differs.

However, Yampa’s signal functions have no effectful interaction: they are Causal Commutative Arrows (Liu, Cheng, Hudak 2009)

\textbf{Both} considered identities actually hold.
Yet another attempt at exercise 2

```haskell
circuit_v3 :: A Double Double
  circuit_v3 = (a1 &&& a3)
    >>> first a2
    >>> arr (uncurry (+))
```
Yet another attempt at exercise 2

circuit_v3 :: A Double Double
circuit_v3 = (a1 &&& a3)
  >>> first a2
  >>> arr (uncurry (+))

Are circuit_v1, circuit_v2, and circuit_v3 all equivalent?
Point-free vs. pointed programming

What we have seen thus far is an example of *point-free* programming: the values being manipulated are not given any names.
What we have seen thus far is an example of *point-free* programming: the values being manipulated are not given any names.

This is often appropriate, especially for small definitions, and it facilitates equational reasoning as shown by Bird & Meertens (Bird 1990).
What we have seen thus far is an example of **point-free** programming: the values being manipulated are not given any names.

This is often appropriate, especially for small definitions, and it facilitates equational reasoning as shown by Bird & Meertens (Bird 1990).

However, large programs are much better expressed in a **pointed** style, where names can be given to values being manipulated.
Ross Paterson’s \texttt{do}-notation for arrows supports \textit{pointed} arrow programming. Only \textit{syntactic sugar}.

\begin{verbatim}
proc \texttt{pat} -> do [ \texttt{rec} ]
\texttt{pat}_1 <- \texttt{sfexp}_1 <- \texttt{exp}_1
\texttt{pat}_2 <- \texttt{sfexp}_2 <- \texttt{exp}_2
\ldots
\texttt{pat}_n <- \texttt{sfexp}_n <- \texttt{exp}_n
\texttt{returnA} <- \texttt{exp}
\end{verbatim}

\textbf{Also:} \texttt{let \texttt{pat} = \texttt{exp} } \equiv \texttt{ pat <- arr \texttt{id} <- \texttt{exp}}
Let us redo exercise 3 using this notation:

circuit_v4 :: A Double Double
circuit_v4 = proc x -> do
  y1 <- a1 <- x
  y2 <- a2 <- y1
  y3 <- a3 <- x
  returnA <- y2 + y3
The arrow do notation (3)

We can also mix and match:

```
circuit_v5 :: A Double Double
circuit_v5 = proc x -> do
  y2 <- a2 <<< a1 --< x
  y3 <- a3 --< x
  returnA --< y2 + y3
```
**Exercise 4:** Describe the following circuit using the arrow do-notation:

\[ a_1, a_2 :: A \text{ Double Double} \]
\[ a_3 :: A \text{ (Double,Double) Double} \]
The arrow $\texttt{do}$ notation (4)

**Exercise 4:** Describe the following circuit using the arrow $\texttt{do}$-notation:

![Circuit Diagram]

\[
\begin{align*}
\text{a1, a2} &:: \text{A Double Double} \\
\text{a3} &:: \text{A (Double,Double) Double}
\end{align*}
\]

**Exercise 5:** As 4, but directly using only the arrow combinators.
circuit = proc x -> do
   rec
      y1 <- a1 <<- x
      y2 <- a2 <<- y1
      y3 <- a3 <<- (x, y)
      let y = y2 + y3
      returnA <<- y
Some basic signal functions (1)

- `identity :: SF a a`
  
  `identity = arr id`
Some basic signal functions (1)

- **identity :: SF a a**
  
  \[
  \text{identity} = \text{arr } \text{id}
  \]

- **constant :: b -> SF a b**
  
  \[
  \text{constant } b = \text{arr } (\text{const } b)
  \]
Some basic signal functions (1)

- **identity** :: SF a a
  \[ \text{identity} = \text{arr id} \]

- **constant** :: b -> SF a b
  \[ \text{constant } b = \text{arr (const } b) \]

- **integral** :: VectorSpace a s=>SF a a
  It is defined through:

  \[ y(t) = \int_{0}^{t} x(\tau) \, d\tau \]
Some basic signal functions (2)

- \( \text{iPre} :: a \rightarrow \text{SF} \ a \ a \)
Some basic signal functions (2)

- \( \text{iPre} :: a \rightarrow \text{SF} \ a \ a \)
- \( (^<<) :: (b \rightarrow c) \rightarrow \text{SF} \ a \ b \rightarrow \text{SF} \ a \ c \) 
  \( f \ (^^<<) \ sf = sf \ \gg \gg \ \text{arr} \ f \)
Some basic signal functions (2)

- \( \text{iPre} :: \text{a} \rightarrow \text{SF a a} \)
- \( (^<<) :: (\text{b} \rightarrow \text{c}) \rightarrow \text{SF a b} \rightarrow \text{SF a c} \)
  \[ f \, (^<<) \, \text{sf} = \text{sf} \, >> > > \, \text{arr} \, f \]
- \( \text{time} :: \text{SF a Time} \)
Some basic signal functions (2)

• iPre :: a -> SF a a

• (^<<) :: (b->c) -> SF a b -> SF a c
  f (^<<) sf = sf >>> arr f

• time :: SF a Time

Quick Exercise: Define time!
Some basic signal functions (2)

- \( \text{iPre} :: a \rightarrow \text{SF} \ a \ a \)
- \( ^{<<} :: (b \rightarrow c) \rightarrow \text{SF} \ a \ b \rightarrow \text{SF} \ a \ c \)
  \( f \ (^{<<}) \ sf = sf \ >>> \ arr \ f \)
- \( \text{time} :: \text{SF} \ a \ \text{Time} \)

Quick Exercise: Define time!

\( \text{time} = \text{constant} \ 1.0 \ >>> \ \text{integral} \)
Some basic signal functions (2)

- **iPre**: \( a \rightarrow SF\ a\ a \)
- **(<<)**: \( (b \rightarrow c) \rightarrow SF\ a\ b \rightarrow SF\ a\ c \)
  
  \( f\ (<<)\ sf = sf \gggg arr\ f \)
- **time**: \( SF\ a\ Time \)

Quick Exercise: Define time!

\[
\text{time} = \text{constant } 1.0 \gggg \text{integral}
\]

Note: there is **no** built-in notion of global time in Yampa: time is always **local**, measured from when a signal function started.
A bouncing ball

\[ y = y_0 + \int v \, dt \]
\[ v = v_0 + \int -9.81 \]

On impact:

\[ v = -v(t^-) \]

(fully elastic collision)
Modelling the bouncing ball: part 1

Free-falling ball:

type Pos = Double

type Vel = Double

fallingBall ::
    Pos -> Vel -> SF () (Pos, Vel)
fallingBall y0 v0 = proc () -> do
    v <- (v0 +) ^<< integral <- -9.81
    y <- (y0 +) ^<< integral <- v
    returnA <- (y, v)
Events

Conceptually, *discrete-time* signals are only defined at discrete points in time, often associated with the occurrence of some *event*. 
Events

Conceptually, *discrete-time* signals are only defined at discrete points in time, often associated with the occurrence of some *event*.

Yampa models discrete-time signals by lifting the *range* of continuous-time signals:

```
data Event a = NoEvent | Event a
```
Events

Conceptually, *discrete-time* signals are only defined at discrete points in time, often associated with the occurrence of some *event*.

Yampa models discrete-time signals by lifting the *range* of continuous-time signals:

\[
data \text{ Event } a = \text{NoEvent} | \text{Event } a
\]

*Discrete-time signal* \( = \text{Signal} (\text{Event } a) \).
Events

Conceptually, *discrete-time* signals are only defined at discrete points in time, often associated with the occurrence of some *event*.

Yampa models discrete-time signals by lifting the *range* of continuous-time signals:

```haskell
data Event a = NoEvent | Event a

Discrete-time signal = Signal (Event α).
```

Associating information with an event occurrence:

```haskell
tag :: Event a -> b -> Event b
```
Some basic event sources

- never :: SF a (Event b)
- now :: b -> SF a (Event b)
- after :: Time -> b -> SF a (Event b)
- repeatedly ::
  Time -> b -> SF a (Event b)
- edge :: SF Bool (Event ())
Stateful event suppression

- notYet :: SF (Event a) (Event a)
- once :: SF (Event a) (Event a)
Modelling the bouncing ball: part 2

Detecting when the ball goes through the floor:

```haskell
fallingBall' :: Pos -> Vel -> SF () ((Pos, Vel), Event (Pos, Vel))
fallingBall' y0 v0 = proc () -> do
  yv@(y, _) <- fallingBall y0 v0 -< ()
  hit <- edge -< y <= 0
  returnA -< (yv, hit `tag` yv)
```

ITU FRP 2010: Lecture 2 – p.50/64
Q: How and when do signal functions “start”?
Q: How and when do signal functions “start”?  
A: • **Switchers** “apply” a signal functions to its input signal at some point in time.
Q: How and when do signal functions “start”?

A: • **Switchers** “apply” a signal function to its input signal at some point in time.
  • This creates a “running” signal function *instance*.
Switching

Q: How and when do signal functions “start”? 

A: • *Switchers* “apply” a signal functions to its input signal at some point in time. 
• This creates a “running” signal function *instance*. 
• The new signal function instance often replaces the previously running instance.
Q: How and when do signal functions “start”?

A: • Switchers “apply” a signal functions to its input signal at some point in time.
  • This creates a “running” signal function instance.
  • The new signal function instance often replaces the previously running instance.

Switchers thus allow systems with varying structure to be described.
The basic switch (1)

Idea:

- Allows one signal function to be replaced by another.
- Switching takes place on the first occurrence of the switching event source.

```plaintext
switch ::
    SF a (b, Event c)
-> (c -> SF a b)
-> SF a b
```
The basic switch (1)

Idea:

• Allows one signal function to be replaced by another.

• Switching takes place on the first occurrence of the switching event source.

```
switch ::

  SF a (b, Event c)
  -> (c -> SF a b)
  -> SF a b
```
The basic switch (1)

Idea:

- Allows one signal function to be replaced by another.
- Switching takes place on the first occurrence of the switching event source.

```
switch ::
    Function yielding SF to switch into
    SF a (b, Event c) -> (c -> SF a b) -> SF a b
```
Exercise 6: Define an event counter `countFrom` using

```
countFrom ::
    Int -> SF (Event a) Int
```

using

```
switch :: SF a (b, Event c)
    -> (c -> SF a b)
    -> SF a b
constant :: b -> SF a b
notYet :: SF (Event a) (Event a)
```

and any other basic combinators you might need.
Solution exercise 6

```
countFrom :: Int -> SF (Event a) Int
countFrom n =
  switch
    (constant n &&& identity
      (const (notYet >>> countFrom (n+1)))))
```
Solution exercise 6

Another version that ignores any event at time 0 also from the very start:

countFrom :: Int -> SF (Event a) Int
countFrom n =
  switch
  (constant n &&& notYet)
  (const (countFrom (n+1)))
Modelling the bouncing ball: part 3

Making the ball bounce:

\[
\text{bouncingBall} :: \text{Pos} \to \text{SF} () (\text{Pos}, \text{Vel}) \\
bouncingBall \ y0 = \text{bbAux} \ y0 \ 0.0 \\
\text{where} \\
\text{bbAux} \ y0 \ v0 = \\
\quad \text{switch (fallingBall' } y0 \ v0) \ \& \ (y,v) \ -> \\
\quad \text{bbAux} \ y \ (-v)
\]
Simulation of bouncing ball
Modelling using impulses (1)

From a modelling perspective, using a device like switch to model the interaction between the ball and the floor may seem rather unnatural.
Modelling using impulses (1)

From a modelling perspective, using a device like switch to model the interaction between the ball and the floor may seem rather unnatural.

A more appropriate account of what is going on is that an impulsive force is acting on the ball for a short time.
From a modelling perspective, using a device like switch to model the interaction between the ball and the floor may seem rather unnatural.

A more appropriate account of what is going on is that an impulsive force is acting on the ball for a short time.

However, Yampa does provide a derived version of integral capturing the basic idea:

```haskell
impulseIntegral ::
    VectorSpace a k =>
    SF (a, Event a) a
```
The decoupled switch

dSwitch ::
   SF a (b, Event c)
   -> (c -> SF a b)
   -> SF a b

• Output at the point of switch is taken from the old subordinate signal function, not the new residual signal function.

• This means the output at the current point in time is independent of whether or not the switching event occurs at that point in time. Hence decoupled.
The recurring switch

$rSwitch, \ drSwitch ::$
\[ SF\ a\ b \to SF\ (a,\ Event\ (SF\ a\ b))\ b \]

- Switching events received on the signal function input, carrying signal function to switch into.
- Switching occurs whenever an event occurs, not just once.
- Can be defined in terms of $\text{switch/dSwitch}$.
Reading (1)


Reading (2)


Reading (3)
